Constraint-based Autonomic Reconfiguration

Hewson, J. A., Anderson, P., & Gordon, A. D. (2013). Constraint-based Autonomic Configuration. In Proceedings of 2013 Self-Adaptive and Self-Organizing systems conference (SASO). IEEE Computer Society. 10.1109/SASO.2013.23

Constraint-Based Autonomic Reconfiguration

John A. Hewson
School of Informatics
University of Edinburgh
john.hewson@ed.ac.uk

Paul Anderson
School of Informatics
University of Edinburgh
dcspaul@ed.ac.uk

Andrew D. Gordon
Microsoft Research &
School of Informatics
University of Edinburgh
adg@microsoft.com

***Abstract*—Declarative, object-oriented configuration management systems are widely used by system administrators. Recently, logical constraints have been added to such systems to facilitate the automatic generation of configurations. However, there is no facility for reasoning about subsequent reconfigurations, such as those needed in an autonomic configuration system. In this paper we develop a number of language primitives, which facilitate not only one-off configuration tasks, but also subsequent reconfigurations in which the previous state of the system is taken into account. We show how it can be directly integrated into a declarative language, and assess its impact on performance.**

I. Introduction

System administrators at large computing installations are increasingly adopting automated tools which make use of declarative, object-oriented languages [1], [2], [3]. These tools replace low-level scripts, which describe the steps needed to achieve a given system state with a declarative model of the goal state of the system. Such tools can be used to configure workstations, servers, and network hardware, as well as application-level services.

Recently, logical constraints have been added to such systems to facilitate the automatic generation of configurations [4], [5], [6], [7]. Such systems are able to perform verification of a system configuration, impact-analysis of configuration changes, generation of valid configurations, and optimisation of a configuration according to some criteria.

However, there is no facility for reasoning about subsequent reconfigurations, such as those needed in an autonomic configuration system. Instead configuration problems are treated as one-off tasks: initial configurations of a new system, starting from scratch. In practice the majority of configuration tasks are incremental, starting from some existing state and applying changes which take the existing configuration into account.

Take for example, the task of assigning virtual machines to physical machines. After a system has been configured initially, it is desirable for subsequent reconfigurations to take into account the current allocation of virtual machines so as not to move virtual machines unnecessarily from one physical machine to the next.

Such unnecessary moves commonly occur when the constraint solver explores a different subsection of the solution space. This is to be expected when simply re-running a modified version of a configuration problem. Thus it is necessary to instead inform the solver about the previous state of the system and how that state affects subsequent configuration decisions.

We wish to develop an autonomic configuration system. In this paper we develop a method for taking into account the previous state of a declarative configuration system. The previous state is made available to the constraint solver, preventing unnecessary changes. External parameters are gathered from the environment. We define primitives for an existing declarative configuration language that express how the state affects subsequent reconfigurations. Finally, we evaluate the performance of our *custom* reconfiguration models against two simpler approaches. First, the *none* strategy; a re-run of the original model altered to reflect the current system state. Second, the *automatic* strategy, in which custom reconfiguration goals are replaced with a heuristic constraint which minimises the number of variables modified by a reconfiguration. We found that the custom reconfiguration models, which use novel reconfiguration primitives, performed significantly better than either of the simpler strategies. We believe that declarative configuration languages would benefit from the adoption of such primitives.

Contributions of the paper

1) Define primitives for expressing reconfiguration constraints in a constraint-based language.
2) Extend a declarative configuration language with state-aware reconfiguration primitives.
3) Define the translation of the state-aware primitives to a constraint program.
4) Show that models which use the reconfiguration primitives can be solved more efficiently.
5) Show that translated models can scale to reconfiguration problems of a useful size.

Structure of the paper

First we introduce a declarative configuration language (II). Next we explain the notion of reconfiguration, our reconfiguration primitives (III). We then describe a method for their transformation into a CSP (IV). Finally, we provide experimental results measuring the effectiveness and performance of reconfiguration constraints (VI).

II. Background

Below we give a short example of a configuration modelled with ConfSolve [4], an object-oriented declarative configuration language, in which logical constraints on a system can be specified and solved as a constraint satisfaction problem (CSP):

```
class Server {
  var hasPublicIP as bool;
}
```

```
class Application {
  var host as ref Server;
}

class Apache extends Application {
  this.hasPublicIP = true;
}

var s1 as Server;
s1.hasPublicIP = true;

var s2 as Server;
s2.hasPublicIP = false;

var apache as Apache;
```

This example shows a simple problem in which there are two servers `s1` and `s2`, the former which has a public IP address. An Apache web server application, `apache`, is to be hosted on one of the servers. The constraint solver will decide which. Apache instances are constrained to be hosted only on servers with public IP addresses. In this case, `apache` will be hosted on `s1`. The decision variable in this example is **`ref`** `host`, in the `Application` class, which is a reference to a `Server` instance.

III. Reconfiguration

In this paper we propose three constraint-based language primitives for reconfiguration:

parameters	are variables which take their value from an external source;
previous value expressions	provide access to the prior solution;
init and change blocks	allow for separate initial and reconfiguration constraints to be specified.

For example, we could use these primitives to express a virtual machine allocation problem in which reconfiguration should never move a virtual machine off its current host, unless that host has failed. For the sake of example, we ignore the constraints which pack virtual machines onto physical hosts:

```
class Machine {
  param failed as bool;
}

abstract class VM {
  var host as ref Machine;
}

var rack1 as Machine[48]; // standard rack size
var vms as VM[192];       // 4*VM per Machine

change {
  forall vm in vms where !vm.host.failed {
    vm.host = ~vm.host;
  };
}
```

The `Machine` class represents a physical machine, which may be in a `failed` state. The **`param`** keyword indicates that the value for this variable is taken from some source external to the model, typically a monitoring system. When the value of a monitored parameter changes, a reconfiguration is triggered. The `VM` class represents a virtual machine, which has a physical `host` reference. Variables `rack1` and `vms` provide instances of both physical and virtual machines. Finally, the **`change`** block contains the constraints which are enforced when a reconfiguration occurs. In this case the **`change`** constraint specifies that if a virtual machine's host has not failed, then the value of `vm.host` must be equal to its value in the previous solution, $\sim$`vm.host`. The operator $\sim e$ yields the value of expression e in the previous solution to the model and may only be used within a **`change`** block. Parameter values are described as CSON (ConfSolve Object Notation) and provided in a separate file. The structure of the parameter file must match the structure of the ConfSolve model, such as in the following extract, in which the second machine in `rack1` has failed:

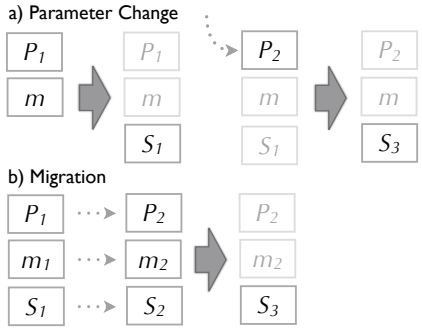


Fig. 1: Parameter Change vs Migration. Bold arrows indicate compilation/solving. **a)** Initial configuration of model m with parameters P_1 results in solution S_1. Updated parameters P_2 are used to find a new solution S_3. **b)** Migration via modification of model m_1 to m_2 along with its parameters and solution is followed by re-solving to get solution S_3

```
rack1: [
  Machine { failed: false },
  Machine { failed: true },
  Machine { failed: false },
  ...
```

A. Parameter changes and migrations

Where m is a model, P are the model's parameters, and S is the model's previous solution, initially undefined. There are two reconfiguration scenarios:

parameter change	is a reconfiguration triggered by a change in value of one of the model's parameters P. An external system monitors and updates parameter values to reflect the current system state.
migration	is a modification of the model m itself, altering its abstract syntax tree. The result is a new model. The model's parameters P and previous solution S may also be correspondingly migrated.

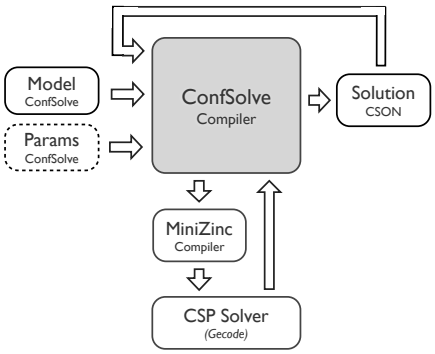


Fig. 2: Compiling and solving a reconfigurable model. White boxes are files; shaded boxes are processes. The *Gecode* CSP solver is used to find solutions to the compiled MiniZinc models.

These two reconfiguration scenarios are illustrated in Figure 1. Both take into account the model m, previous solution S, and parameters P. Thus when reconfiguring the next solution S' is a function of m, S, and P.

Thus a *parameter change* is explicitly captured in the model: the parameters which are expected to change regularly, perhaps every few minutes. A *migration* is a less-common alteration of the model itself.

Migrations include changes to the infrastructure or the kinds of services present, *i.e.*, the removal and addition of new classes, objects and constraints. Migrations are always manual edits to the original model. Certain migrations can be handled automatically by the compiler, without the user needing to update the state to correspond to the new model. These migrations are ones in which there are only deletions or additions to the model, such as increasing the cardinality of a set of objects, or deleting a variable declaration. Renaming a variable is an example of a migration which does not fit this pattern, and would instead require a manual editing of not only the model, but also the current recorded state.

IV. Implementation

We have implemented our reconfiguration primitives as an extension of the ConfSolve configuration system [4]. The ConfSolve compiler translates an object-oriented configuration model into MiniZinc, a simple logical language for expressing constraint problems. The MiniZinc model is then solved using Gecode [8], a state-of-the-art CSP solver. Figure 2 details the architecture, in which both parameters and the previous solution are stored in CSON and passed to the ConfSolve compiler, in addition to the existing model.

V. Translation to MiniZinc

An example ConfSolve to MiniZinc translation is given below. The model contains four object instances of the class `Server` which has a solver-assigned id between 1 and 10. A reconfiguration constraint restricts the value of `id` to be equal to its previous value (if any):

```
class Server {
  var id as 1..10;
  change { id = ~id; }
}
var servers as Server[4];
```

The MiniZinc translation of this problem introduces the prior state in a variable prefixed with `old_` and translates the reconfiguration constraint accordingly. The individual fields are flattened into arrays, indexed by object. The object indices are in the range 1..4 as there are four instances of `Server` in the model:

```
// MiniZinc
1..4: servers = [1,2,3,4];
array[1..4] of var 1..10: Server_id;
array[1..4] of 1..10: old_Server_id =
                                    [2,4,6,8];
constraint
  forall (i in 1..4) (
    Server_id[i] = old_Server_id[i];
  );
```

A. Translation

We combine our reconfiguration primitives with the ConfSolve language, and provide a formal description of their translation to MiniZinc.

Syntax of Types:

$T ::=$	type
bool	boolean
int	integer
c	object
$T[\,]$	set of T
$c\,[n]$	set of objects, with cardinality n

The syntax of types for ConfSolve. Identifiers are represented by metavariables: c is a class name, v is a variable name, l is a field name; n is an integer; and b is a boolean.

Syntax of CSON values:

$V ::=$	value
i	integer
true | **false**	boolean
c {Member*}	object
ref Target	object reference
$T[n]\{V_1, \ldots, V_n\}$	set literal
Member ::=	member
$v : V$	variable name : value
Target ::=	target
v	variable
Target.l	field access
Target[i]	set access

The translation steps below make use of ConfSolve's types and values, the abstract syntax as described in [4] and reproduced above.

Translation begins with *static allocation*, in which indices are assigned to each object, and the upper bound on the number of instances count(c) is calculated for each class c.

Translation of CSON values $[\![V]\!]$:

Given a CSON value V, its translation to MiniZinc is:

when $V =$ **true** $\vee$ $V =$ **false** $\vee$ $V = i$

V

when $V = c$ {Member*} $\vee$ $V = c[n]\{V_1, \ldots, V_n\}$

undefined

when $V = T[n]\{V_1, \ldots, V_n\}$

{ for $V_i \in \{V_1, \ldots, V_n\}$, $[\![V_i]\!]$ }

when $V =$ **ref** Target

the index of the object O where path(O) = Target

The MiniZinc translation $[\![V]\!]$ of a ConfSolve value V is used throughout the following translations. Boolean and integer values are translated as literals. The translation of objects and sets of objects is undefined, this is firstly because their MiniZinc representation consists of a constant integer index (or a set of these), which is already know from the static allocation phase described previously. Secondly, the expansion of the object's fields into arrays is handled later as part of the translation of variables, so there is not always a one to one correspondence between CSON values and MiniZinc values.

The translation of an object reference is the index of the object in the model whose path matches the Target expression, which may be either a variable, a field access, or an indexed object-set access. Formally, we define it as the object O with path(O) = Target where path(O) is the CSON Target expression corresponding to the full path of the object within the global scope. Each object has a unique Target path, for example `rack[2].machines[4].hostname`.

Parameters

The values of all ConfSolve parameters are contained in a single CSON abstract syntax tree (AST), which follows the same structure as the original model. Parameter values must match their declared type. These do not affect the object counting performed in the static allocation phase, because it is based on the type only and not the value. Sets of objects already have a fixed cardinality as part of their type, so there is no way to introduce extra objects.

The translation of parameters is described as two rules, which correspond to class-level and global scopes. These rules make use of the *corresponding CSON value* for a given parameter, that is, the CSON value which has the equivalent path in the CSON AST as the given parameter. Such a CSON value must exist for a every parameter.

Translation of global parameter declarations:

For each global declaration **param** v **as** T, and corresponding CSON value V, introduce a declaration:

when $T = c[n] \vee T = c$

the translation of global **var** v **as** T, from [4]

otherwise

$[\![T]\!] : v = [\![V]\!]$

For each global declaration **param** v **as ref** c, and corresponding CSON value V, introduce a declaration:

$[\![c]\!] : v = [\![V]\!]$

The translation of objects and sets of objects remains unchanged. This is because they are translated to constant object indices, generated during the static allocation phase. Thus the purpose of an object parameter is not to specify the object id, but to allow the values of its fields to be assigned values.

For all other types, parameters are translated into MiniZinc constants, which consist of a compound declaration and assignment. Reference parameters are translated in a similar manner, in which the CSON path to the object is mapped into an object index via $[\![V]\!]$.

Translation of class-level parameter declarations:

For each declaration of some class c where $\text{count}(c) > 0$, containing fields **param** f_i **as** T_i $^{i \in 1..n}$, and corresponding CSON values V_j $^{j \in 1..n}$, introduce a declaration for each field f_i:

when $T = c[n] \vee T = c$

the translation of class-level **var** v **as** T, from [4].

otherwise

array [1 **..** $\text{count}(c)$**] of** $[\![T_i]\!] : c\_f_i =$

[for $j \in 1..\text{count}(c), [\![V_j]\!]$ **]**

Where class c contains fields **param** f_i **as ref** c_i $^{i \in 1..n}$, introduce a declaration for each field f_i:

array [1 **..** $\text{count}(c)$**] of** $[\![c_i]\!] : c\_f_i =$ (as above)

Variables nested within classes are translated using an **array** containing all instances of a particular field, where the object index is the index into the array. Objects and sets of objects are once again translated in the same manner as **var** declarations in standard MiniZinc. All other types are translated as an **array** for each field, containing the translated CSON value for each object index in $1..\text{count}(c)$. Reference parameters are translated in the same manner.

Previous values & change expressions

The previous solution to a model is represented as a CSON AST. This is the output of a previous run of ConfSolve. As with parameters, the values in the CSON solution does not affect the object counting performed in the static allocation phase, because it is based on type only, and not on value. The number of objects is therefore fixed.

We assume that the AST of the model has not changed since it was used to generate the previous solution, however the value of the parameters may change freely. We also include some relaxations of this assumption to simplify *migrations*.

Translation of change and init blocks:

Iff performing a reconfiguration, for each expression block **change** $\{e_1 \ldots e_n\}$, introduce a expression:

$e_1 \wedge \cdots \wedge e_n$

Iff performing an initial configuration, for each expression block **init** $\{e_1 \ldots e_n\}$, introduce a expression:

$e_1 \wedge \cdots \wedge e_n$

Change and init blocks are expression blocks containing constraints, and can appear both at the global level and nested

within classes. Their translation is a simple form of conditional compilation: change blocks are translated as the conjunction of their constraint expressions only when a re-configuration is being performed, otherwise they are not translated. Likewise for init blocks.

Translation of global variable declarations:

For each global declaration **var** v **as** T, where $T \neq c[n] \wedge T \neq c$, and corresponding CSON value V, introduce a declaration:

$[\![T]\!]$: `old_`$v = [\![V]\!]$

For each global declaration **var** v **as ref** c, introduce a declaration:

$[\![c]\!]$: `old_`$v = [\![V]\!]$

For global variables which are not objects or sets of objects, a MiniZinc constant is introduced with the appropriate type, and with value equal to the MiniZinc translation of its corresponding CSON value. The MiniZinc variable name is prefixed with `old_` as its purpose is to expose the previous value of that variable within the MiniZinc model. Reference variables undergo an equivalent translation.

Translation of class-level variable declarations:

For each declaration of some class c where $\mathrm{count}(c) > 0$, containing fields **var** f_i **as** T_i $^{i \in 1..n}$ and corresponding CSON values V_j $^{j \in 1..n}$, where $T_i \neq c[n] \wedge T_i \neq c$, introduce a declaration for each field f_i:

array [1 **..** $\mathrm{count}(c)$] **of** $[\![T_i]\!]$: `old_`c`_`$f_i =$
[for $j \in 1..\mathrm{count}(c), [\![V_j]\!]$]

If any corresponding CSON value V_j is undefined, then substitute the anonymous variable _ in place of $[\![V_j]\!]$ and introduce the constraint:
constraint $c\_f_i[j] =$ `old_`c`_`$f_i[j]$

Where class c contains fields **var** f_i **as ref** c_i $^{i \in 1..n}$, introduce a declaration for each field f_i:

array [1 **..** $\mathrm{count}(c)$] **of** $[\![c_i]\!]$: `old_`c`_`f_i = (as above)

An equivalent translation step is introduced for class-level variables, resulting in the introduction of `old_` prefixed MiniZinc variables. Variables nested within classes are translated using an **array** containing all instances of a particular field, where the object index is the index into the array. Variables which declare objects or sets of objects do not have a translation, because their translation consists of constant object indexes generated in the static allocation phase, as was the case with the translation of parameters.

There is a special provision made for when the CSON value V_i corresponding to field f_i is not present in the previous solution's CSON AST. This is to support migrations in which variables declaring objects are added and is the only situation in which this can occur. For example, in a migration in which a new object is added to an existing model for which a previous solution already exists. In this case we first attempt to find the corresponding CSON value in the previous solution by traversing its AST. When this fails, MiniZinc's anonymous variable _ is used in its place and a constraint is introduced requiring the new and old values of the variable to be equivalent.

Translation of expressions $[\![e]\!]$**:**

$v \triangleq [\![v]\!]$

$$e.l \triangleq \begin{cases} \texttt{old\_classof}(e)\texttt{\_}l\,[\![e]\!] & \text{if } e.l \text{ is a sub-expression of some } {\sim}e' \\ \mathrm{classof}(e)\texttt{\_}l\,[\![e]\!] & \text{otherwise} \end{cases}$$

${\sim}e \triangleq [\![e]\!]$

The previous value expression $\sim e$ is defined as simply $[\![e]\!]$, but the rules for both variable (v) translation and member expression $e.l$ translation are made context-dependent to make use of previous values. In the translation of member expressions, there are now two cases. The second case is the standard translation which is an array access where $[\![e]\!]$ is the object index. The first case applies whenever the member expression is a sub-expression of a previous-value expression and results in the same translation except that the resulting MiniZinc variable receives the `old_` prefix. The mechanism for accessing previous values can now be seen, it is enough to prefix any variable with `old_` to access its prior value which was introduced during the translation of variable declarations.

Translation of variables $[\![v]\!]$**:**

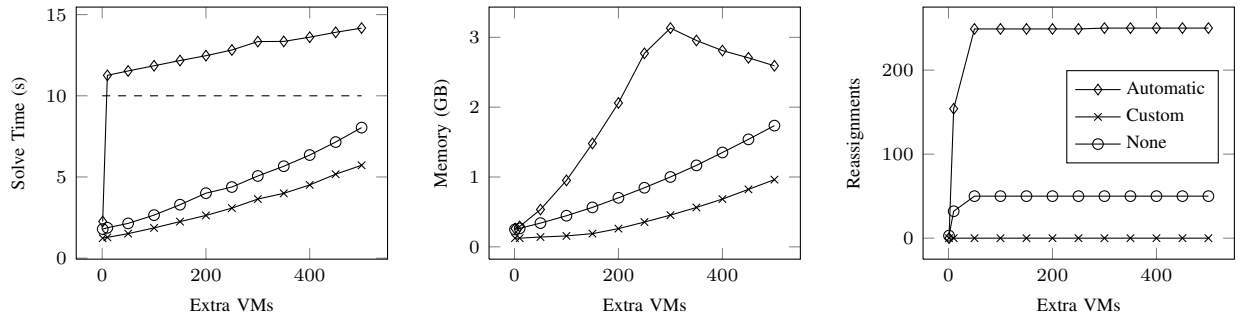
Within the scope of class c, the translation $[\![v]\!]$ of a variable v is:

when v is a sub-expression of some $\sim e$, and v is not quantified:
  when v is declared in class $c' \in c^*$
    `old_`c'`_`v[this]
  otherwise
    `old_`v
otherwise
  when v is declared in class $c' \in c^*$
    c'`_`v[this]
  otherwise
    v

Where c^* is the set containing c and all its ancestors.

The translation of variables undergoes a similar modification as that of member expressions. When a variable is a sub-expression of a previous-value expression $\sim e$ then its standard translation is prefixed with `old_`. The standard translation has two cases; the first for class-level variables, and the second for global variables.

We impose an additional restriction on when variables within a previous-value sub-expression are translated with the `old_` prefix. If a variable is quantified, *i.e.*was introduced by a Fold expression, such as x in the expression **forall** $(x$ **in** $e_1)$ (e_2), then it is not translated in this manner. This is because quantified variables are bound to the expression over which they quantify, thus it is not meaningful to talk about the previous value of a quantified variable such as $\sim x$, instead we must quantify over $\sim e_1$. This approach has the benefit of not adding any further complexity to the translation of folds.



(a) (b) (c)

Fig. 3: Adding Virtual Machines. **a)** Solve time, with timeout at 10 seconds indicated by the dashed line. Values above timeout are due to time spent parsing FlatZinc. The *automatic* strategy quickly times-out. **b)** Solver memory usage, with high memory usage for the *automatic* strategy. **c)** Solution quality, with 250 reassignments for the automatic strategy, and 50 for *none*.

VI. Evaluation

In order to evaluate our implementation we examine both the quality of solutions and the impact of reconfiguration on performance. We compare three reconfiguration strategies:

none	ignores the previous system state
custom	uses a model's custom `change` expressions, to access previous system state
automatic	uses a simple heuristic in place of a model's `change` expressions

The *automatic* heuristic introduces a **change** constraint for each variable: **maximize bool2int**$(v = \sim v)$. This allows us to measure the effectiveness of having custom **change** constraints as a language feature, rather than having them determined by the compiler automatically.

We examine three reconfiguration scenarios, which represent the three modes in which ConfSolve can operate: a migration, a parameterised model, and a migration of a parameterised model.

Experimental Setup: The evaluation was performed on a machine with a 2.5GHz Intel Core 2 Quad processor and 8GB of RAM, running Ubuntu 12.04. We used the 64-bit MiniZinc to FlatZinc converter version 1.6.0 with the `--no-optimize` flag, and the 64-bit Gecode FlatZinc interpreter version 3.7.3.

A. Migration: Adding Virtual Machines

We use ConfSolve to generate an assignment of virtual machines to physical machines in an Infrastructure as a Service (IaaS) configuration, and then add more virtual machines as a migration.

Each physical machine is identical, having 8 CPUs and 16GB or memory. Each virtual machine has variables representing its requirements on the physical machine resources. These are declared as follows:

```
class Machine {
  var cpu as int = 16;        // 8-core
  var memory as int = 16384; // 16GB
  var disk as int = 2048;     // 2TB
}
```

```
class VM {
  var host as ref Machine;
  var cpu as int = 1;
  var memory as int = 768;
  var disk as int = 20;
}
```

The infrastructure consists of two racks of 48 physical machines, onto which we wish to allocate 250 virtual machines:

```
var machines as Machine[48];
var vms as VM[250];
```

A bin-packing constraint on virtual machine placement prevents over-provisioning of host resources, *i.e.*, for each physical machine the sum of required resources must be less than the amount provided by the machine:

```
forall m in machines {
    sum vm in vms where vm.host = m {
      vm.cpu;
    } <= m.cpu
    && sum vm in vms where vm.host = m {
      vm.memory;
    } <= m.memory
    && sum vm in vms where vm.host = m {
      vm.disk;
    } <= m.disk;
};
```

Finally, a reconfiguration constraint, stating that each virtual machine should remain on its previous host:

```
change {
  forall vm in vms { vm.host = ~vm.host; };
}
```

To perform the evaluation, the size of the virtual machine set `VM[250]` is incrementally increased by editing the model file. The results of scaling this problem up to 500 virtual machines are shown in Figure 3. The custom **change** expressions outperform both the *automatic* and *none* strategies, with regard to both time and memory. The automatic approach quickly reaches the solver timeout of 10 seconds before completing its search, though it is still able to produce sub-optimal results as it progresses.

The quality of the solutions follows a similar trend, with the custom **change** expressions performing a perfect reconfiguration with no reassignments of existing machines. The *automatic* strategy quickly tends towards 250 reassignments, the maximum possible. The *none* strategy levels-off at 50 reassignments, which reflects the default behaviour of the Gecode solver.

Result: For this use case, the addition of custom **change** expressions to ConfSolve results in both an increase in time and in particular memory performance of the solver, and successfully prevents unnecessary configuration changes when compared with the *none* and *automatic* strategies.

B. Parameters: Virtual Server Failure

We use ConfSolve to generate an assignment of virtual machines to physical machines in an Infrastructure as a Service (IaaS) configuration, and then fail some of the machines via a parameter.

Each physical machine is identical, having 2 CPUs and 2GB or memory. A parameter `online` indicates whether a machine is online or has failed:

```
class Machine {
  param online as bool;
  var cpu as int = 2;
  var memory as int = 2048;
  var disk as int = 10;
}
```

Each virtual machine is identical and has variables representing its requirements on the physical machine resources:

```
class VM {
  var host as ref Machine;
  var cpu as int = 1;
  var memory as int = 1024;
  var disk as int = 5;
}
```

The infrastructure consists of 200 physical machines, onto which we wish to allocate 200 virtual machines, with a 2:1 ratio this means that the physical machines are at 50% capacity:

```
var machines as Machine[200];
var vms as VM[200];
```

A bin packing constraint identical to that in section VI-A is added to the model, which we do not show here. This ensures that physical machines are not over-provisioned.

Virtual machines are constrained to be hosted only on machines which are online:

```
forall vm in vms {
  vm.host.online = true;
};
```

We wish to distribute the virtual machines across the infrastructure, leaving headroom, rather than packing them tightly on to physical machines. As this is a preference we make use of a soft constraint, to minimise the number of virtual machines with a common host:

```
minimize sum vm1 in vms {
  count (vm2 in vms
         where vm1.host = vm2.host);
};
```

Finally, a reconfiguration constraint, which requires each virtual machine to remain on its previous host as long as that host was previously online and is so currently. The intent of this constraint is the same as for the previous example, the additional complexity comes from the need to take into account machine failure:

```
change {
  forall m in machines
         where m.online && ~m.online {
    forall vm in vms
           where ~vm.host = m {
      vm.host = ~vm.host;
    };
  };
}
```

To perform the evaluation, the sizes of both the set of physical machines and the set of virtual machines are simultaneously increased in size, while maintaining their ratios. An initial configuration is performed, followed by a reconfiguration in which 50% of the machines have their `online` parameter set to false.

The results of scaling this problem up to 300 virtual machines are shown in Figure 4. Custom **change** expressions narrowly outperform the *automatic* and *none* strategies with regard to time, diverge towards significant improvements with regard to memory from around 150 machines.

The quality of the solutions forms two distinct categories. The custom **change** expressions perform a perfect reconfiguration, in which only 50% of the virtual machines are reassigned. The *automatic* and *none* strategies both perform the maximum possible number of reassignments, 100%.

Result: For this use case, custom **change** expressions are a valuable addition to ConfSolve, resulting in a minimal reconfiguration, where a maximal one would otherwise have occurred.

C. Migration with Parameters: Cloudbursting

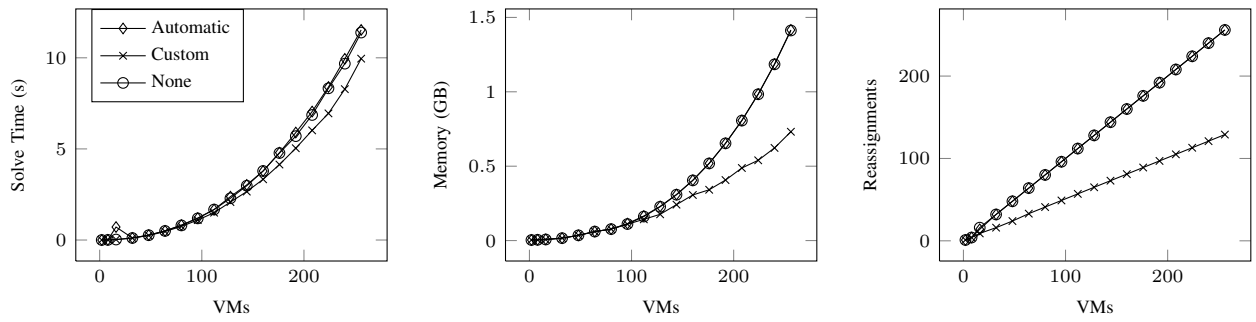
This evaluation combines a migration with parameter changes, in order to show that both can occur simultaneously. We model a scenario known as *cloudbursting* in which excess load from an enterprise datacenter may be run on the cloud.

We create an abstract class to represent a host, which may be either a physical machine with 4 CPUs and 4GB of RAM, or a cloud, which has no fixed resources. Physical machines have an `online` parameter:

```
abstract class Host {}

class Machine extends Host {
  param online as bool;
  var cpu as int = 4;
  var memory as int = 4096;
}

class Cloud extends Host {}
```



(a) (b) (c)

Fig. 4: Virtual Server Failure. **a)** Solve time, showing a marginal advantage to the *custom* strategy. **b)** Solver memory usage, with a significant advantage to the *custom* strategy. **c)** Solution quality, showing *custom* achieving 50% reassignments, the minimum possible.

Services are tasks which can be placed on hosts, and have requirements on the amount of CPU and memory required to run them:

```
abstract class Service {
  var host as ref Host;
  var cpu as int;
  var memory as int;
}
```

We define three specific types of task: web, worker, and database, with differing CPU and memory requirements:

```
class Web extends Service {
  cpu = 2; memory = 2048; }

class Worker extends Service {
  cpu = 2; memory = 2048; }

class Database extends Service {
  cpu = 4; memory = 4096; }
```

The infrastructure consists of 300 physical machines within the enterprise, and a single cloud provider:

```
var enterprise as Machine[300];
var cloud as Cloud;
```

We create web, worker, and database services in the ratio 2:2:1.

```
var webs as Web[200];
var workers as Worker[200];
var databases as Database[100];
```

So that we may more easily quantify over all services, a set of `service` references is created, which the solver will automatically resolve to the declarations above:

```
var services as ref Service[500];
```

A bin-packing constraint for the services hosted in the enterprise should be familiar from the previous examples:

```
forall m in enterprise {
  sum s in services where s.host = m {
    s.cpu;
  } <= m.cpu
  &&
  sum s in services where s.host = m {
    s.memory;
  } <= m.memory;
};
```

We do not wish to host services in the cloud if there is available capacity within the enterprise. The constraint below states that if the number of services hosted in the cloud is greater than zero, then the number of services hosted in the enterprise is equal to the number of enterprise machines which are online. The `->` operator represents logical implication:

```
count (s in services
       where s.host = cloud) > 0 ->
  count (s in services
         where s.host in enterprise) =
           count (m in enterprise
                  where m.online);
```

We constrain services to be placed only on machines which are online:

```
forall s in services {
  s.host.online = true;
};
```

Finally, there is a reconfiguration constraint, similar to that from section VI-B, except that `machines` is substituted for `enterprise` and `vms` for `services`. This requires each service to remain on its previous host as long as that host was previously online and is so currently. The constraint applies only to machines within the enterprise, not the cloud.

To perform the evaluation, an initial configuration is performed, after which the number of `Worker` services is doubled by manually editing the model, as a migration. Additionally, 50% of the machines have their `online` parameter set to false.

The results of scaling the problem up to 300 machines are shown in Figure 5. With regard to time, custom **change** expressions narrowly outperform the *none* strategy, while the *automatic* strategy tends rapidly towards a solver timeout at 60 seconds. In terms of memory performance, custom **change** expressions significantly outperform both of the other strategies, showing much better scaling.

The quality of the solutions follow a new pattern. Both the *custom* and *automatic* strategies achieve a perfect reconfiguration, in which only 50% of the services are reassigned. Though the automatic strategy yields worse results after it starts to time-out at 250 machines. The *none* strategy remains poor, performing the maximum number of reassignments, 100%.

Result: For this use case, custom **change** expressions show their value in terms of performance, even though the *automatic* strategy is able to provide results of the same quality. In practice, this problem is most likely to be memory-bound, thus the *custom* strategy offers desirable benefits.

VII. Related Work

A. Reconfiguration

Engage [9] is a prototype deployment system which makes use of the MiniSat SAT solver to solve dependency constraints between components. It has a small, formalised type system with component subtyping. It does not include algebraic constraints.

The 2011 Google/ROADEF challenge [10] covered a machine reassignment task at large-scale. It was won by a custom-coded local search algorithm, which was able to find high-quality solutions within five minutes.

The bin-repacking scheduling problem was studied in [11] and the results applied to the virtual machine manager Entropy. A custom CSP model of the problem was built, and incorporated into Entropy via an open-source CSP solver, which performed well.

In [12], the Prolog-based DALI multi-agent system is combined with Lira, a network-based reconfiguration system. The combined system allows global reconfigurations to be performed dynamically through the cooperation of the agents. Reconfiguration tasks are encoded as a set of action rules with preconditions. This differs significantly from the declarative approach taken by ConfSolve, in which only the goal state, and not the steps taken to achieve it, are of concern.

Dynamic Software Updating (DSU) [13], in which C programs are updated at runtime, are of relevance to ConfSolve. This is due to DSU's ability to infer a patch based on changes to a source file, in a manner similar to the Unix `diff` utility. Some changes ultimately require manual intervention, but the approach taken to automating this process may be applicable to ConfSolve's migrations.

Planit [14] combines a simple system configuration tool with the LPG planner to perform reconfiguration tasks. It has a built-in model of machines and components which may run upon them. Reconfiguration is performed after component failure by incorporating the non-failed components into the goal state used by the planner, and updating the initial state to match the system. This means that reconfigurations which would require moving a non-failed component are not possible.

The SmartFrog and LCFG configurations tools were combined in [15] to create a prototype tool capable of exploiting SmartFrog's component-based peer-to-peer orchestration with LCFG's low-level system configuring abilities. Although the system is able to respond to change, its logic to do so is custom-coded in Java for each component.

In [16] a graph-based language for reconfiguring of software architectures is proposed. Reconfigurations are treated as operations on graphs, namely additions and removal of components and connections. The language is procedural, with scripts specifying a series of actions on components, with preconditions.

B. Configuration

Declarative configuration tools such as CFEngine [1] have not traditionally included constraints. Early experimenters with constraints adopted Prolog for this purpose [17], followed [18], and more recently Yin [19].

The Alloy Analyzer [20] is a SAT-based modelling system which shares some commonality with ConfSolve: both provide an object-oriented specification language with logical constraints, and both require the user to specify an upper-bound on the number of objects in the search space. However, Alloy's generality goes beyond that of ConfSolve as a system for model-checking stateful systems. It is unsuitable as a backend for ConfSolve, in place of MiniZinc, because of its lack of optimisation constraints.

A key influence on ConfSolve was Cauldron [5], an object-oriented configuration language based on the CIM [21] model of classes, object references, and arrays. Solutions are generated using the VeriFun theorem prover, which itself relies on a SAT solver. Unlike ConfSolve, the Cauldron language is not rigorously defined, whether or not its search is complete is unclear, and its translation to SAT is not disclosed. Furthermore, its prototype implementation does not scale well to problems beyond tens of machines, as we discovered in [4].

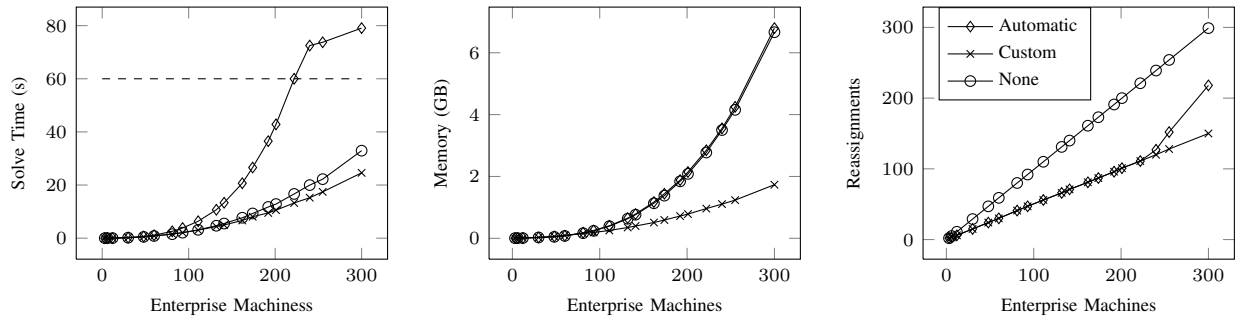
We defined the ConfSolve language in [4], and evaluated its performance against a virtual infrastructure management problem. An executable semantics in the form of a formalised translation to MiniZinc is given. This initial version of ConfSolve, like its predecessors, performed one-off configuration tasks only.

VIII. Conclusion and Future Work

Reconfiguration is an important part of the configuration process, and incorporating it into a constraint-based generative configuration system is non-trivial. We have proposed new features for configuration languages to better achieve this, and shown that state can be incorporated in a declarative manner compatible with existing tools.

Our reconfiguration primitives are compatible with existing object-oriented configuration languages which feature logical constraints. By describing our primitives using MiniZinc we provide an executable semantics which can scale to non-trivial problem sizes using an off-the-shelf CSP solver. Our evaluation shows that common virtual machine reconfigurations perform better with our reconfiguration primitives than without.

Future work in this area could take advantage of the different performance profiles of different solvers, or attempt to produce optimised MiniZinc constraints. There is scope for expanding support for automated migrations, and identifying when an inferred source change can be safely applied to a new model.



(a) (b) (c)

Fig. 5: Cloudbursting. **a)** Solve time, showing the *automatic* strategy timing-out at 60 seconds. **b)** Solver memory usage, with a significant advantage to the *custom* strategy. **c)** Solution quality, showing *custom* and *automatic* achieving 50% reassignments, the minimum possible, until *automatic* times-out at around 225 machines.

The ConfSolve compiler (v0.7) is written in OCaml and is available from http://homepages.inf.ed.ac.uk/s0968244/confsolve.

ACKNOWLEDGEMENTS

This work was funded by Microsoft Research through their European PhD Scholarship Programme.

REFERENCES

[1] M. Burgess *et al.*, "CFEngine: a site configuration engine," *USENIX Computing systems*, vol. 8, no. 3, pp. 309–402, 1995.

[2] Puppet Labs, "Puppet," 2008, available from http://www.puppetlabs.com/puppet/.

[3] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft, "The SmartFrog configuration management framework," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 16–25, January 2009. [Online]. Available: http://doi.acm.org/10.1145/1496909.1496915

[4] J. Hewson, P. Anderson, and A. D. Gordon, "A declarative approach to automated configuration," in *26th Large Installation System Administration Conference (LISA'12)*, 2012.

[5] L. Ramshaw, A. Sahai, J. Saxe, and S. Singhal, "Cauldron: A policy-based design tool," in *7th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2006)*, 2006, pp. 113–122.

[6] S. Narain, "Network configuration management via model finding," in *Proceedings of the 19th conference on Large Installation System Administration Conference*. USENIX Association, 2005, p. 15.

[7] A. Sahai, S. Singhal, R. Joshi, and V. Machiraju, "Automated generation of resource configurations through policies," in *IEEE Policy*, 2004.

[8] Gecode Team. (2006) Gecode: Genetic constraint development environment. Available from http://www.gecode.org.

[9] J. Fischer, R. Majumdar, and S. Esmaeilsabzali, "Engage: a deployment management system," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 263–274.

[10] ROADEF. (2011) Google ROADEF/EURO challenge 2011-2012: Machine reassignment. [Online]. Available: http://challenge.roadef.org/2012/files/problem_definition_v1.pdf

[11] F. Hermenier, S. Demassey, and X. Lorca, "Bin repacking scheduling in virtualized datacenters," in *Principles and Practice of Constraint Programming – CP 2011*. Springer, 2011.

[12] M. Castaldi, S. Costantini, S. Gentile, and A. Tocchio, "A logic-based infrastructure for reconfiguring applications," in *Declarative Agent Languages and Technologies*, ser. Lecture Notes in Computer Science, J. Leite, A. Omicini, L. Sterling, and P. Torroni, Eds. Springer, 2004, vol. 2990, pp. 17–36.

[13] M. Hicks and S. Nettles, "Dynamic software updating," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 6, pp. 1049–1096, Nov. 2005.

[14] N. Arshad, D. Heimbigner, and A. Wolf, "Deployment and dynamic reconfiguration planning for distributed software systems," in *Tools with Artificial Intelligence, 2003. Proceedings. 15th IEEE International Conference on*, 2003, pp. 39–46.

[15] P. Anderson, P. Goldsack, and J. Paterson, "SmartFrog meets LCFG: Autonomous reconfiguration with central policy control," in *Proceedings of the 17th conference on Large Installation System Administration Conference*, 2003, pp. 219–228.

[16] M. Wermelinger, A. Lopes, and J. L. Fiadeiro, "A graph based architectural (re)configuration language," in *Proceedings of the 8th European software engineering conference*, ser. ESEC/FSE-9. New York, NY, USA: ACM, 2001, pp. 21–32.

[17] A. Couch and M. Gilfix, "It's elementary, dear Watson: applying logic programming to convergent system management processes," in *Proc. LISA '99*. USENIX, 1999.

[18] S. Narain, T. Cheng, B. Coan, V. Kaul, K. Parmeswaran, and W. Stephens, "Building autonomic systems via configuration," in *Proceedings of IEEE Autonomic Computing Workshop*, 2003.

[19] Q. Yin, J. Cappos, A. Baumann, and T. Roscoe, "Dependable self-hosting distributed systems using constraints," in *Proceedings of the Fourth conference on Hot topics in system dependability*. USENIX Association, 2008, pp. 11–11.

[20] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, 2002.

[21] Distributed Management Task Force. (2010) Common information model (CIM) standards. Available from http://www.dmtf.org/standards/cim/. [Online]. Available: http://www.dmtf.org/standards/cim/